3 Transformationen

Häufig ist es sinnvoll, die Koordinaten der Punkte, die die zu zeichnenden Elemente definieren, vor dem Zeichnen einer Transformation (Verschiebung, Drehung, Spiegelung, Skalierung) zu unterwerfen. So kann man die Darstellung eines Objekts in einer speziellen Lage des Koordinatensystems programmieren und die Koordinaten vor der tatsächlichen Zeichenaktion transformieren. Besonders erleichtert wird so auch die mehrfache Darstellung des gleichen Objekts in unterschiedlichen Lagen.

Die Koordinaten von dreidimensionalen Objekten müssen in jedem Fall vor der graphischen Darstellung auf eine zweidimensionale Ebene projiziert werden. Deshalb werden Beispiele zu dreidimensionalen Problemen erst im Kapitel 4 behandelt.

Die Lösung beider Probleme (Transformation bzw. Projektion) wird von Methoden der Klasee **CPT** unterstützt, die in den Kapiteln 3 und 4 beschrieben werden. Vorangestellt werden jeweils die theoretischen Grundlagen, die als Basis für die **CPT**-Methoden dienen.

3.1 Homogene Koordinaten

Speziell für die Probleme der projektiven Geometrie wird mit erheblichem Vorteil die Darstellung eines Punktes im Raum durch vier Angaben (an Stelle der drei kartesischen Koordinaten) beschrieben. Diese sogenannten **homogenen Koordiaten** stehen mit den kartesischen Koordinaten in einem einfachen Zusammenhang:

$$\text{Homogene Koordinaten} \quad \rightarrow \quad \begin{bmatrix} x \\ y \\ z \\ \lambda \end{bmatrix} \quad \Rightarrow \quad \begin{bmatrix} x/\lambda \\ y/\lambda \\ z/\lambda \end{bmatrix} \quad \leftarrow \quad \text{Kartesische Koordinaten}$$

Dabei kann λ einen beliebigen Wert ungleich Null haben, für den Spezialfall $\lambda = \mathbf{1}$ sind die ersten drei Komponenten in der Darstellung eines Punktes mit homogenen Koordinaten mit den kartesischen Koordinaten identisch.

Der Vorteil, der sich aus der Verwendung homogener Koordinaten ergibt, wird sich in der Möglichkeit der einheitlichen Darstellung unterschiedlicher Transformationen zeigen, wodurch sich nacheinander auszuführende Transformationen mathematisch sehr übersichtlich verknüpfen lassen. Bei der Projektion von Raumpunkten in eine Darstellungsebene ergeben sich die Abbildungen automatisch in (ebenen) homogenen Koordinaten.

Folgende Vorstellung kann mit der sicher etwas ungewöhnlichen Beschreibung eines Punktes durch vier Angaben verknüpft werden: Der darzustellende Punkt definiert gemeinsam mit dem Nullpunkt des Koordinatensystems eine Gerade. Wenn λ die Werte von $-\infty$ bis $+\infty$ durchläuft, dann werden alle Punkte auf dieser Geraden beschrieben (und im Vorgriff auf das Thema "Projektion": Wenn diese Gerade die "Blickrichtung" definiert, werden alle Punkte der Geraden auf den gleichen Punkt der "Projektionsebene" abgebildet). Aber selbst für die ebenen Probleme allein ergeben sich Vorteile durch die Verwendung homogener Koordinaten.

3.2 Ebene Transformationen

Zwei verschiedene Transformationen werden betrachtet:

- Wenn sich ein Objekt bezüglich eines **festen Koordinatensystems** bewegt, dann ändern sich die Koordinaten seiner Punkte nach den Regeln der **geometrischen Transformation**.
- Wenn das Koordinatensystem selbst bewegt wird, dann ändern sich die Koordinaten der Punkte eines sich nicht mitbewegenden Objekts nach den Regeln der **Koordinatentransformation**.

Es genügt, das "Schicksal" eines Punktes bei einer Transformation zu untersuchen. In allen Fällen wird im folgenden die "alte" Lage des Punktes durch die Koordinaten x und y beschrieben, die Lage nach der Transformation durch die Koordinaten x' und y'.

Es werden folgende vier sehr einfache Transformationen beschrieben (Translation, Rotation um den Nullpunkt, Skalierung bezüglich des Nullpunktes, Spiegelung an den Koordinatenachsen), aus denen sich durch geeignete Verknüpfungen beliebige andere Transformationen zusammensetzen lassen:

- Bei einer **Translation** bewegen sich alle Punkte des Objekts (geometrische Transformation) bzw. des Koordinatensystems (Koordinatentransformation) auf kongruenten Bahnen. Bei der **geometrischen Translation** mögen die Koordinaten aller Punkte des Objekts die Veränderungen t_x und t_y erfahren, so daß die neue Lage eines Punktes durch

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

  beschrieben wird.

- Bei einer **Rotation** drehen sich das Objekt (geometrische Transformation) bzw. das Koordinatensystem (Koordinatentransformation) um einen bestimmten Winkel **um eine vorzugebende Achse**, beim ebenen Problem also um einen vorzugebenden Punkt. Mit einer kleinen Skizze macht man sich leicht klar, daß bei einer **Rotation eines Punktes um den Nullpunkt (geometrische Rotation)** um den (entgegen dem Uhrzeigersinn positiv gezählten) Winkel φ folgende Transformation gilt:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x\cos\varphi - y\sin\varphi \\ x\sin\varphi + y\cos\varphi \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad .$$

- Bei der **Skalierung** werden die Abmessungen des Objekts (geometrische Transformation) bzw. die Skaleneinteilung der Koordinatenachsen (Koordinatentransformation) vergrößert (Skalierungsfaktoren größer als 1) bzw. verkleinert. Die Skalierung bezieht sich immer auf einen zu definierenden Punkt, der dann selbst seine Lage nicht

verändert, während alle anderen Punkte ihren Abstand vom Bezugspunkt vergrößern oder verkleinern. Bei der **geometrischen Skalierung bezüglich des Nullpunktes** mit den Skalierungsfaktoren s_x und s_y (jeweils in Richtung der Koordinatenachsen) berechnet sich die Lage des neuen Punktes nach

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad .$$

- Bei einer **Spiegelung eines Objektes an der y-Achse (geometrische Transformation)** ändern alle x-Koordinaten der Punkte ihr Vorzeichen:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -x \\ y \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad .$$

Dementsprechend gilt für die **Spiegelung eines Objektes an der x-Achse (geometrische Transformation)**:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ -y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad .$$

Da sich die Translation als einzige Transformation in kartesischen Koordinaten nicht durch eine Transformationsmatrix beschreiben läßt, kann hier erstmals mit Vorteil von den homogenen Koordinaten Gebrauch gemacht werden. Die kartesischen Koordinaten werden um $\lambda = 1$ ergänzt, und alle Transformationen lassen sich einheitlich durch Transformationsmatrizen beschreiben. Während für die Translation das Aufschreiben der Beziehung als Multiplikation "Matrix * Vektor" überhaupt erst möglich wird, werden die Transformationsmatrizen für die Rotation, die Skalierung und die Spiegelung um eine einfache Zeile bzw. Spalte ergänzt ("gerändert"). Diese Formeln werden nachfolgend zusammengestellt.

3.2.1 Ebene geometrische Transformation mit homogenen Koordinaten

Die Formeln beschreiben die **Bewegung eines Punktes im festen Koordinatensystem!**

Translation um t_x und t_y:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{T}_G \, \bar{x}$$

Rotation um den Nullpunkt mit dem Winkel φ entgegen dem Uhrzeigersinn:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{R}_G \, \bar{x}$$

Skalierung bezüglich des Nullpunktes um s_x und s_y:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{S}_G \, \bar{x}$$

Spiegelung an der *y*-Achse:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{y,G} \, \bar{x}$$

Spiegelung an der *x*-Achse:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{x,G} \, \bar{x}$$

3.2.2 Ebene Koordinatentransformation mit homogenen Koordinaten

Die Formeln gelten bei **Bewegung des Koordinatensystems (Objekte bleiben in Ruhe)!**

Translation um t_x und t_y:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{T}_K \, \bar{x} = \bar{T}_G^{-1} \, \bar{x}$$

Rotation um den Nullpunkt mit dem Winkel φ entgegen dem Uhrzeigersinn:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{R}_K \, \bar{x} = \bar{R}_G^{-1} \, \bar{x}$$

Skalierung der Einheiten bezüglich des Nullpunktes um s_x und s_y:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{S}_K \, \bar{x} = \bar{S}_G^{-1} \, \bar{x}$$

Spiegelung an der *y*-Achse:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{y,K} \, \bar{x} = \bar{M}_{y,G}^{-1} \, \bar{x}$$

Spiegelung an der *x*-Achse:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{M}_{x,K} \, \bar{x} = \bar{M}_{x,G}^{-1} \, \bar{x}$$

Die angegebenen Formeln zur Koordinatentransformation leiten sich aus ähnlich einfachen Überlegungen her, wie sie für die geometrischen Transformationen diskutiert wurden. Bemerkenswert ist, daß alle Transformationsmatrizen der Koordinatentransformationen die Inversen zu den entsprechenden Transformationsmatrizen der geometrischen Transformationen sind (und natürlich umgekehrt). Dies ist allerdings leicht einzusehen, weil eine Koordinatentransformation mit den gleichen Parametern wie eine entsprechende geometrische Transformation ein "Nachführen des Koordinatensystems" bedeutet, so daß der ursprüngliche Zustand wieder hergestellt wird.

In der **CPT**-Klasse sind nur Methoden verfügbar, die die **geometrischen Transformationen** realisieren, nach den oben angegebenen Formeln sind sie allerdings leicht auch für eventuell auszuführende Koordinatentransformationen nutzbar.

3.2.3 Verknüpfung von Transformationen

Aus den Elementartransformationen, für die in den Abschnitten 3.2.1 und 3.2.2 die Formeln angegeben wurden, können beliebige Transformationen durch das Ausführen von mehreren Transformationen nacheinander realisiert werden. Dabei ist zu beachten, daß eine **Transformation immer durch Linksmultiplikation mit der entsprechenden Transformationsmatrix realisiert wird**, so daß schließlich ein Produkt mehrerer Transformationsmatrizen die Gesamt-Transformation beschreibt, bei der die Matrix der letzten Transformation links steht.

Dies soll an einem einfachen Beispiel gezeigt werden: Es ist die geometrische Transformationsmatrix zu bestimmen, mit der die Rotation eines Punktes (x,y) um einen Winkel φ (im Uhrzeigersinn) um den **beliebigen Drehpunkt** (x_M,y_M) erzeugt wird. Da in den Elementartransformationen nur der Fall "Rotation um den Nullpunkt" vorgesehen ist, wird folgendermaßen vorgegangen:

Zunächst wird der Ursprung des Koordinatensystems in den Punkt (x_M,y_M) verschoben (**Koordinatentranslation 1** mit $t_x = x_M$ und $t_y = y_M$), danach kann die **geometrische Rotation** um den Nullpunkt ausgeführt werden, schließlich muß die Verschiebung des Koordinatensystems rückgängig gemacht werden (entweder durch **Koordinatentranslation 2** mit $t_x = -x_M$ und $t_y = -y_M$ oder durch **geometrische Translation** mit $t_x = x_M$ und $t_y = y_M$). Folgende Matrizen sind also zu multiplizieren (man beachte, daß in der Matrix für die Koordinatentranslation die Elemente $-t_x$ und $-t_y$ stehen):

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \bar{T}_{K,2}\ \bar{R}_G\ \bar{T}_{K,1}\ \bar{x} = \bar{T}_{Gesamt} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \bar{T}_{Gesamt}\ \bar{x}$$

mit

$$\bar{T}_{Gesamt} = \begin{bmatrix} 1 & 0 & x_M \\ 0 & 1 & y_M \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_M \\ 0 & 1 & -y_M \\ 0 & 0 & 1 \end{bmatrix} .$$

Man beachte, daß eine Vertauschung der Reihenfolge zweier Transformationen in der Regel auf eine ganz andere Gesamt-Transformation führt. In der mathematischen Beschreibung verknüpfter Transformationen mit homogenen Koordinaten wird dies durch die Matrizen-Produkte der Transformationsmatrizen deutlich: Im Gegensatz zum Produkt skalarer Größen ist das Matrix-Produkt nicht kommutativ.

Die Klasse **CPT** enthält je eine Transformationsmatrix für die ebene Transformation und eine Transformationsmatrix für die 3D-Transformation. Beide Matrizen werden (im **CPT**-Konstruktor) als Einheitsmatrizen initialisiert, so daß zunächst keine Unterschiede bei der Verwendung der zeichnenden Funktionen, die die Transformation auswerten, und den bisher behandelten Zeichenfunktionen bestehen. Erst dann, wenn mit den entsprechenden **CPT**-Methoden eine Transformation geändert wurde, macht sich dies (nur) bei den speziellen Zeichenfunktionen bemerkbar.

Die Zeichenfunktionen, die die Transformation auswerten, gehören (wie die bisher behandelten Zeichenfunktionen) zur Klasse **CGI**. Da die Klasse **CGI** aber von der Klasse **CPT** abgeleitet ist, enthält sie auch die Transformationsmatrizen, und ihre Instanzen können alle **CPT**-Methoden benutzen.

Dies wird im Abschnitt 3.4 für ein ebenes Problem demonstriert. Da für räumliche Probleme neben der (optionalen) Transformation immer eine Projektion erforderlich ist, um das 3D-Objekt in der ebenen Zeichenfläche darzustellen, werden im folgenden Abschnitt die 3D-Transformationsformeln nur zusammengestellt. Ihre Verwendung wird erst im Kapitel 4 nach der Behandlung der Projektionen an Beispielen demonstriert.

3.3 3D-Transformationen

Nachfolgend werden die Transformationsformeln für die geometrische Transformation und die Koordinatentransformation eines 3D-Punktes angegeben. Es gelten alle Aussagen, die im Abschnitt 3.2 (ebene Transformationen) gemacht wurden, insbesondere die Aussagen zur zusammengesetzten Transformation, zusätzlich ist zu beachten:

- Die Rotation bedeutet im Raum immer "Rotation um eine vorzugebende Achse" (das gilt natürlich auch für die Ebene, dort ist von der Rotationsachse aber immer nur ein Punkt zu sehen). Als "elementare Rotationen" werden die Rotationen um die drei Koordinatenachsen betrachtet (im Gegensatz zu einer "Elementar-Rotation" in der Ebene, der Rotation um den Nullpunkt). Rotationen um beliebige Achsen sind durch "Verknüpfung von Transformationen" (vgl. Abschnitt 3.2.3) zu realisieren.
- Spiegelung ist im Raum "Spiegelung an einer Ebene", Elementartransformationen spiegeln an ***x*-*y***-Ebene, ***x*-*z***-Ebene und ***y*-*z***-Ebene. Die besonders einfachen Transformationsmatrizen dafür werden nachfolgend nicht mit angegeben. Sie unterscheiden sich von der Einheitsmatrix jeweils um genau ein Minuszeichen auf der Hauptdiagonalen.

3.3.1 Geometrische 3D-Transformation mit homogenen Koordinaten

Translation um t_x, t_y und t_z:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{T}_G \, \bar{x}$$

Skalierung bezüglich des Nullpunktes um s_x, s_y und s_z:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{S}_G \, \bar{x}$$

Rotation um die x-Achse mit dem Winkel φ_x:

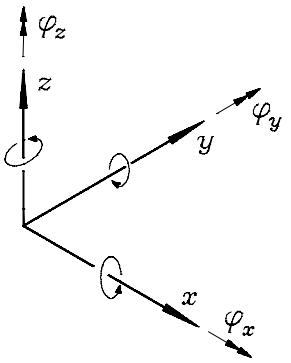
$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi_x & -\sin\varphi_x & 0 \\ 0 & \sin\varphi_x & \cos\varphi_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Gx} \, \bar{x}$$

Rotation um die y-Achse mit dem Winkel φ_y:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi_y & 0 & \sin\varphi_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi_y & 0 & \cos\varphi_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Gy} \, \bar{x}$$

Rotation um die z-Achse mit dem Winkel φ_z:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi_z & -\sin\varphi_z & 0 & 0 \\ \sin\varphi_z & \cos\varphi_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Gz} \, \bar{x}$$



Definition positiver Drehwinkel

Mit diesen Formeln wird die **Bewegung eines Punktes im festen Koordinatensystem** beschrieben.

3.3.2 3D-Koordinatentransformation mit homogenen Koordinaten

Translation um t_x, t_y und t_z:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{T}_K \, \bar{x}$$

Skalierung bezüglich des Nullpunktes um s_x, s_y und s_z:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{S}_K \, \bar{x}$$

Rotation um die x-Achse mit dem Winkel φ_x:

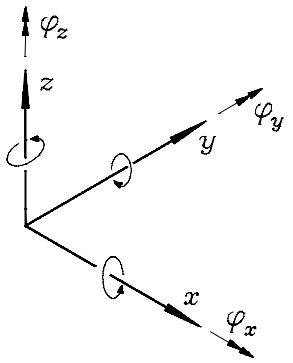
$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi_x & \sin\varphi_x & 0 \\ 0 & -\sin\varphi_x & \cos\varphi_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Kx} \, \bar{x}$$

Rotation um die y-Achse mit dem Winkel φ_y:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi_y & 0 & -\sin\varphi_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi_y & 0 & \cos\varphi_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Ky} \, \bar{x}$$

Rotation um die z-Achse mit dem Winkel φ_z:

$$\bar{x}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\varphi_z & \sin\varphi_z & 0 & 0 \\ -\sin\varphi_z & \cos\varphi_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \bar{R}_{Kz} \, \bar{x}$$



Definition positiver Drehwinkel

Mit diesen Formeln werden die Veränderungen der Koordinaten eines festen Punktes bei **Bewegung des Koordinatensystem** be schrieben.

3.4 Die "t...-Funktionen" (2D) in den Klassen CPT und CGI

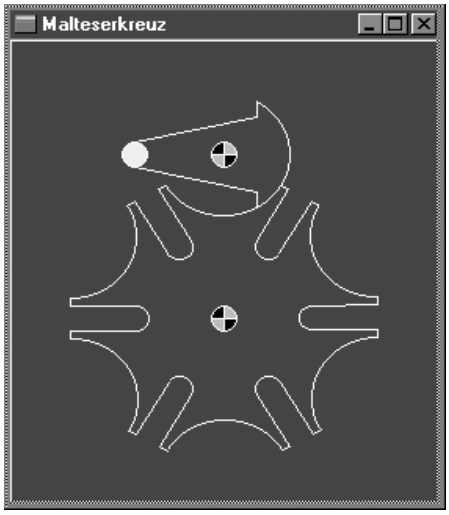
Die sogenannten **"t...-Funktionen"** (alle zugehörigen Funktionsnamen beginnen mit **t**, die erst im folgenden Kapitel behandelten 3D-Funktionen beginnen mit **t3**) lassen sich in zwei Gruppen einteilen:

- Die **"vorbereitenden t...-Funktionen"** gehören zur Klasse **CPT**. Sie definieren bzw. ändern die gültige ebene Transformationsmatrix.
- Die **"zeichnenden t...-Funktionen"** gehören zur Klasse **CGI**. Sie arbeiten wie die im Abschnitt 2.2 beschriebenen "u...-Funktionen" mit "User coordinates", beziehen sich auf das gültige (mit **stuca_gi** bzw. **stuci_gi** eingestellte) Koordinatensystem, wenden aber vor der eigentlichen Zeichenaktion die gültige ebene Transformationsmatrix auf die übergebenen Koordinaten an.

Nicht zu allen "u...-Funktionen" existieren auch die entsprechenden "zeichnenden t...-Funktionen", konsequent kann das Prinzip der Vorab-Transformation ohnehin nur für die Argumente realisiert werden, die Punkt-Koordinaten beschreiben ("Move to", "Line to", "Polygon"). Die "t...-Funktionen", mit denen Kreise bzw. Kreisbögen mit Mittelpunkt und Radius beschrieben werden, wenden auf den Radius z. B. nur die eingestellte Skalierung an.

Nachfolgend werden das Prinzip des Arbeitens mit den "t...-Funktionen" und die Verwendung der wichtigsten Funktionen am Beispiel des Programms **malteser.cpp** (Zeichnen eines symbolischen Malteserkreuz-Getriebes) dargestellt, das schrittweise entwickelt wird.

Die nebenstehende Skizze zeigt das Schema des Getriebes (dieses Bild wird schließlich vom Programm **malteser.cpp** erzeugt), das nach folgendem Prinzip arbeitet. Die "Kurbel" (im Bild oben) dreht sich normalerweise mit konstanter Winkelgeschwindigkeit. Der Stift am Ende der Kurbel greift bei jeder Umdrehung in einen Schlitz des "Malteserkreuzes" ein und dreht dieses weiter. Wenn der Stift den Schlitz wieder verläßt, kommt das Kreuz zum Stillstand, bis der Stift bei der folgenden Kurbelumdrehung in den nächsten Schlitz eingreift.

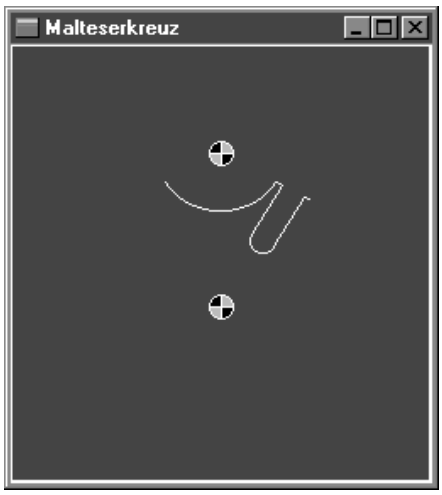


Schema eines Malteserkreuz-Getriebes

Das Malteserkreuz der dargestellten Getriebe-Variante besteht aus 6 jeweils um 60° versetzten "Schwalbenschwänzen", die jeweils durch einen Schlitz mit halbkreisförmigem Grund getrennt sind. Es bietet sich also an, nur ein Sechstel des Bildes zu programmieren und mit geeigneten Rotationstransformationen wiederholt zu erzeugen. Die Abbildung auf der folgenden Seite oben rechts zeigt das zu programmierende Sechstel (und die beiden Drehzapfen), dieses Bild wird von der Version **maltesr0.cpp** erzeugt. Schon für das Erzeugen dieses Teilbildes werden mit Vorteil Rotationstransformationen eingesetzt:

- Vor der ersten Verwendung einer "zeichnenden t...-Funktion" muß die ebene Transformationsmatrix initialisiert werden. Dies geschieht natürlich im Konstruktor von **CPT**. Wenn allerdings (wie im Programm **malteser.cpp**, der Grund dafür wird später

ersichtlich) eine "langlebende" **CGI**-Instanz verwendet wird (vgl. Abschnitt 2.3.1), die über die gesamte Lebensdauer des Hauptfensters existieren soll, muß sie mit **CPT::tinit_pt** bei jedem **OnPaint**-Aufruf erneut initialisiert werden. Damit wird eine Einheitsmatrix vorgegeben (keine Transformation), so daß die "zeichnenden t...-Funktionen" sich zunächst wie die entsprechenden "u...-Funktionen" verhalten.



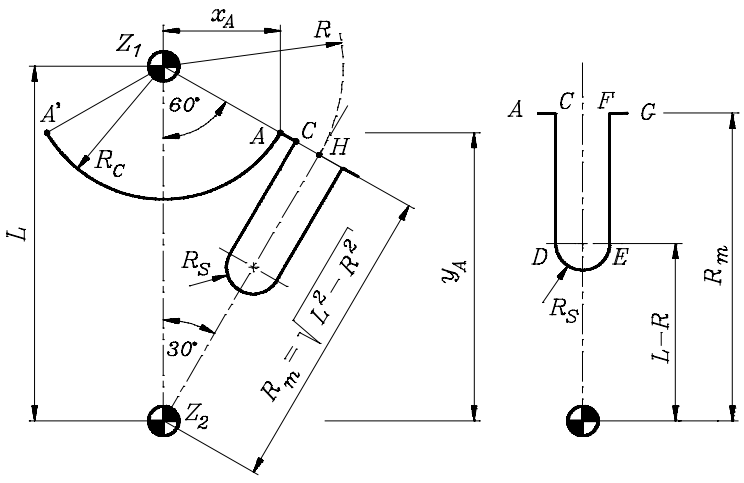
Ein Sechstel des Malteserkreuzes

- Die **CPT**-Methode **trota_pt** realisiert die (im Abschnitt 3.2.3 als Beispiel demonstrierte) "Rotation um einen vorzugebenden Punkt". **Sie arbeitet inkrementell**, fügt also die Rotation (durch entsprechende Matrix-Multiplikation) einer bereits eingestellten Transformation hinzu. Sie ist in **cgiw.h** mit folgendem Prototyp vertreten:

```
void  trota_pt  (double xm , double ym , double phi) ;
```

Anzugeben sind also die "User coordinates" des Punktes, um den gedreht wird, der letzte Parameter ist der Winkel für die Rotation (positiv entgegen dem Uhrzeigersinn, Bogenmaß!).

Um den nachfolgenden Ausschnitt aus dem Programmcode von **maltesr0.c** besser zu verstehen, zeigt die nebenstehende Skizze die typischen Abmessungen, die die Geometrie des Malteserkreuzes bestimmen:



Die Geometrie des Malteserkreuzes, der Schlitz wird in der rechts dargestellten vertikalen Lage programmiert

Die Mittelpunkte der beiden Drehzapfen $\boldsymbol{Z_1}$ und $\boldsymbol{Z_2}$ bilden mit dem Punkt $\boldsymbol{H}$ ein rechtwinkliges Dreieck. Die eingezeichneten Winkel und die Abmessungen $\boldsymbol{L}$, $\boldsymbol{R}$, $\boldsymbol{R_C}$ und $\boldsymbol{R_S}$ definieren die aus Kreisbögen und Geraden aufgebaute Kontur.

Die Abmessungen werden als "private"-Daten in der Klasse **CMainFrame** untergebracht, sie erhalten ihre Werte im Konstruktor **CMainFrame::CMainFrame**. Zur Klasse **CMainFrame** gehört auch das **CGI**-Objekt ("langlebendes" Objekt) **m_gi**, das bei der Konstruktion der Hauptfenster-Klasse auch konstruiert wird, aber natürlich noch keinen Pointer auf einen "Device context" enthalten kann.

Die Zeichenaktion wird bei der Bearbeitung der Botschaft WM_PAINT in **CMainFrame::OnPaint** ausgeführt:

```
void CMainFrame::OnPaint ()
{
   CPaintDC  dc (this) ;
   m_gi.init_gi (this , &dc) ;
   m_gi.vback_gi (CBrush (m_gi.mkrgb_gi (m_gi.GI_BLUE)) ,
                  CPen   (PS_SOLID , 1 , m_gi.mkrgb_gi (m_gi.GI_WHITE))) ;
   m_gi.stuci_gi (- m_L , - m_L , m_L , m_L + m_R + m_Rs , 5.) ;
   m_gi.tinit_pt () ;                          // Transformation initialisieren
   DrawCross (CPen (PS_SOLID , 1 , m_gi.mkrgb_gi (m_gi.GI_WHITE))) ;
                                               // ... zeichnet das Malteserkreuz
   DrawPivot (0.)   ;                          // Lagersymbole
   DrawPivot (m_L)  ;                          // zeichnen
   m_gi.remdc_gi () ;
}
```

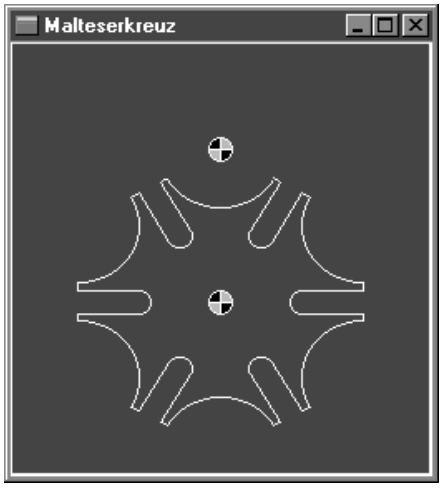
Zunächst wird das **CGI**-Objekt mit **CGI::init_gi** initialisiert. Mit **CGI::vback_gi** wird ein blauer Hintergrund erzeugt, mit **stuci_gi** werden **isotrope** "User coordinates" passend zu den maximalen Abmessungen bei Einhaltung eines kleinen Randes eingestellt (der Koordinatenursprung liegt in der Mitte des unteren Drehzapfens). Vor dem Aufruf von **DrawCross** (Zeichnen des Malteserkreuzes) wird mit **tinit_pt** die ebene Transformationsmatrix initialisiert (Einheitsmatrix = keine Transformation).

Mit dem Aufruf von **DrawPivot** wird jeweils ein Drehzapfen gezeichnet. Dafür werden ausschließlich "u...-Funktionen" verwendet, die die eingestellte Transformation nicht berücksichtigen. Die eigentlich interessante Zeichenaktion wird durch eine Sequenz von "t...-Funktionen" in **DrawCross** realisiert (der "Pen", mit dem gezeichnet werden soll, wird als Referenz erwartet):

```
void CMainFrame::DrawCross (CPen &pen)
{
   CDC* dc_p = m_gi.getdc_gi () ;
   CPen*  PenOld_p  = dc_p->SelectObject (&pen) ;
// for (int i = 0 ; i < 6 ; i++)
   {
      m_gi.tdarc_gi (0. , m_L , m_Rc , - m_xA , m_yA , m_xA , m_yA) ;
      m_gi.tmove_gi (m_xA , m_yA) ;
      m_gi.trota_pt (0. , 0. , - GI_PI / 6.) ;        // ... um 30° drehen
      m_gi.tline_gi (- m_Rs , m_Rm) ;
      m_gi.tline_gi (- m_Rs , m_L - m_R) ;
      m_gi.tdarc_gi (0. , m_L - m_R , m_Rs , - m_Rs ,
                          m_L - m_R , m_Rs , m_L - m_R) ;
      m_gi.tmove_gi (m_Rs , m_L - m_R) ;
      m_gi.tline_gi (m_Rs , m_Rm) ;
      m_gi.trota_pt (0. , 0. , - GI_PI / 6.)  ;       // ... noch einmal 30°
      m_gi.tline_gi (- m_xA , m_yA) ;
   }
   dc_p->SelectObject (PenOld_p)   ;
}
```

- Mit der **CGI-inline**-Methode **getdc_gi** wird der für das Einsetzen des "Pens" erforderliche Pointer auf den "Device context" besorgt.
- Mit **CPT::tdarc_gi** wird der Kreisbogen ***A'-A*** mit dem Mittelpunkt bei **(0,m_L)** gezeichnet, weil noch keine Transformation eingestellt wurde, genau mit den angegebenen Parametern. Danach wird der Punkt ***A*** mit **CGI::tmove_gi** als "Current position" festgelegt (Ausgangspunkt für das Zeichnen der Geraden ***A-C***).

- Bevor die Gerade ***A*-*C*** gezeichnet wird, stellt **CPT::trota_pt** eine Rotation um 30° um den Nullpunkt ein (negativer Winkel, weil in Uhrzeigerrichtung gedreht wird). Damit können alle Koordinaten für das Zeichnen des Schlitzes bis zum Punkt ***F*** der wesentlich einfacheren vertikalen Lage (wie eingangs in der bemaßten Skizze rechts dargestellt) entnommen werden.
- Das Zeichnen der letzten Geraden ***F*-*G*** würde die (etwas umständliche) Berechnung der Länge dieses Geradenstücks erfordern. Da wir uns schon vor der Berechnung der Länge des Geradenstücks ***A*-*C*** "gedrückt" haben, bietet sich der gleiche Trick hier ein zweites Mal an: Mit **CPT::trota_pt** wird "um 30° weitergedreht" (jetzt wird also eine 60°-Rotation vor dem Zeichnen ausgeführt, weil **CPT::trota_pt** inkrementell arbeitet), und für den Zielpunkt des letzten Geradenstücks werden die Koordinaten des Punktes ***A'*** noch einmal verwendet, denn bei ***G*** muß sich ja das nächste Sechstel anschließen.

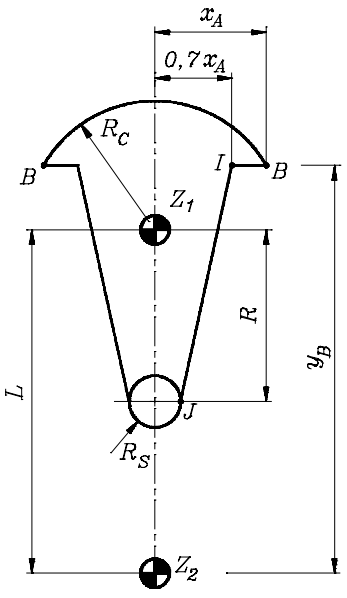


Das komplette Malteserkreuz

Da die gesamte eingestellte Transformation jetzt eine 60°-Rotation um den Nullpunkt ist, kann sich unmittelbar das Zeichnen des nächsten Sechstels **mit exakt den gleichen Funktionen, aufgerufen mit unveränderten Argumenten,** anschließen.

Die oben angegebene Sequenz von Zeichenbefehlen muß also nur in eine sechsfach abzuarbeitende Schleife gelegt werden, um das komplette Malteserkreuz zu erzeugen. In **maltesr0.c** sind diese Zeilen bereits vorgesehen, es müssen nur die vor dem Schleifenkopf stehenden Kommentarsymbole entfernt werden. Danach erzeugt das Programm das nebenstehend zu sehende Bild des kompletten Malteserkreuzes.

Die Programmversion **maltesr1.c** zeichnet (neben den beiden Drehzapfen zur Orientierung) nur die Kurbel, deren Abmessungen aus der nebenstehenden Skizze zu entnehmen sind. Die Bearbeitung der Botschaft WM_PAINT ist nur in einer Zeile gegenüber der Programmversion **maltesr0.c** geändert: An Stelle von **DrawCross** wird die Funktion **DrawCrank** aufgerufen.



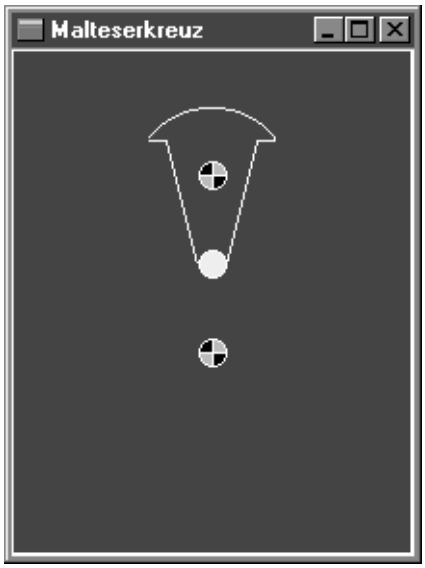
Geometrie der Kurbel

Natürlich wird die Zeichnung in **DrawCrank** in der dargestellten vertikalen Stellung programmiert. Um die "t...-Funktion" **CPT::tmiry_pt** zu demonstrieren, werden die beiden Geradenstücke, die rechts bzw. links spiegelbildlich paarweise auftreten, in einer zweimal zu durchlaufenden Schleife angeordnet, an deren Ende die Transformation "Spiegeln an der ***y***-Achse" eingeschaltet wird. Es soll mit dieser Programmversion auch demonstriert werden, daß die Kurbel trotzdem in einer gedrehten Lage dargestellt werden kann.

Die Funktion **CMainFrame::DrawCrank** erwartet zwei Referenz-Parameter, einen "Pen" für das Zeichnen der Linien und einen "Brush", mit dem der Kreis gefüllt wird, der den Mitnehmer symbolisiert:

```
void CMainFrame::DrawCrank (CPen &pen , CBrush &brush)
{
   CDC* dc_p = m_gi.getdc_gi () ;
   CPen*   PenOld_p   = dc_p->SelectObject (&pen)   ;
   CBrush* BrushOld_p = dc_p->SelectObject (&brush) ;
   for (int i = 0 ; i < 2 ; i++)
   {
      m_gi.tmove_gi (m_xA      , m_yB) ;
      m_gi.tline_gi (m_xA * .7 , m_yB) ;
      m_gi.tline_gi (m_Rs , m_L - m_R) ;
//    m_gi.trota_pt (0. , m_L , - m_OmT) ;   // Bis zur y-Achse drehen
      m_gi.tmiry_pt () ;                     // Spiegeln an der y-Achse
//    m_gi.trota_pt (0. , m_L ,   m_OmT) ;   // Zurueckdrehen
   }
   m_gi.tdarc_gi (0. , m_L , m_Rc , m_xA , m_yB , - m_xA , m_yB) ;
   m_gi.tfcir_gi (0. , m_L - m_R  , m_Rs) ;
   dc_p->SelectObject (PenOld_p)   ;
   dc_p->SelectObject (BrushOld_p) ;
}
```

Die nebenstehende Abbildung zeigt das Bild, das mit **maltesr1.c** erzeugt wird.



Programmierte Kurbelstellung

Wenn die **gesamte Kurbel** einer Transformation unterworfen werden soll (um sie in einer beliebigen gedrehten Lage darzustellen), funktioniert natürlich die einfache Transformation "Spiegeln an der ***y***-Achse" nicht mehr. Dann muß die gleiche Strategie verwendet werden, die im Abschnitt 3.2.3 schon an einem anderen Beispiel demonstriert wurde, hier: "Transformieren, so daß wieder die ***y***-Achse die Spiegelachse ist" ---> "Spiegeln" ---> "Rücktransformieren". In **maltesr1.c** sind die Anweisungen ("herauskommentiert") dafür bereits in **OnPaint** und in **DrawCrank** vorgesehen:

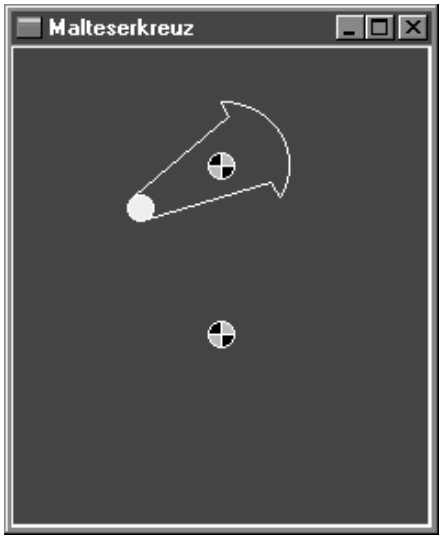
Bei der Bearbeitung von WM_PAINT (in **OnPaint**) wird eine Rotationstransformation (60° im Urzeigersinn) um den Drehpunkt der Kurbel vor dem Aufruf von **DrawCrank** eingefügt:

```
m_gi.m_tinit_gi () ;                        // Initialisieren und ...
m_OmT = - GI_PI / 3 ;
m_gi.trota_gi (0. , m_L , m_OmT) ;          // Setzen der Transformation
DrawCrank (CPen   (PS_SOLID , 1 , m_gi.mkrgb_gi (m_gi.GI_WHITE)) ,
           CBrush (m_gi.mkrgb_gi (m_gi.GI_CYAN))) ;  // Kurbel zeichnen
```

Die Winkel-Variable **m_OmT** gehört zur Klasse **CMainFrame** und ist damit auch in **DrawCrank** verfügbar, so daß die Transformationssequenz dort aktiviert werden kann:

```
m_gi.trota_gi (0. , m_L , - m_OmT) ;          // Bis zur y-Achse drehen
m_gi.tmiry_gi () ;                            // Spiegeln an der y-Achse
m_gi.trota_gi (0. , m_L ,   m_OmT) ;          // Zurueckdrehen
```

Die nebenstehende Abbildung zeigt die Darstellung der Kurbel nach dieser kleinen Änderung.



Kurbel in transformierter Lage

Die so programmierte Darstellung funktioniert natürlich für jeden beliebigen Winkel. Dies wird in der Version **maltesr2.c** ausgenutzt, um zu zeigen, wie man eine einfache Animation mit den "t...-Funktionen" realisieren kann. Dabei wird folgende Strategie verfolgt:

In **CMainFrame::OnCreate** wird ein "Timer" angefordert (und in **CMainFrame::OnClose** wird er wieder freigegeben), der mit "50/1000 Sekunden" auf das kürzeste mögliche Intervall zur Erzeugung von WM_TIMER-Botschaften eingestellt wird (Wert wird automatisch auf 54,925/1000 Sekunden korrigiert). Das Programm erhält also maximal 18,2 Botschaften WM_TIMER pro Sekunde, das ist weniger als beim Kinofilm, aber immerhin wird der Eindruck der Bewegung damit vermittelt. Als Winkelinkrement wird (willkürlich) **m_dOmT = π/15** (**m_dOmT** gehört zur Klasse **CMainFrame**) festgelegt, das sind 12° (wenn es der PC schafft, müßte die Kurbel mit etwa 36,4 Umdrehungen pro Minute rotieren).

Um den Aufwand etwas zu reduzieren, werden die immer wieder benötigten "Pens" und "Brushes" einmal erzeugt und erst beim Schließen des Hauptfensters wieder freigegeben. Im Konstruktor **CMainFrame::CMainFrame** findet man die folgenden zusätzlichen Zeilen:

```
m_penblue   = new CPen   (PS_SOLID , 1 , m_gi.mkrgb_gi (m_gi.GI_BLUE))  ;
m_penwhite  = new CPen   (PS_SOLID , 1 , m_gi.mkrgb_gi (m_gi.GI_WHITE)) ;
m_brushblue = new CBrush (m_gi.mkrgb_gi (m_gi.GI_BLUE)) ;
m_brushcyan = new CBrush (m_gi.mkrgb_gi (m_gi.GI_CYAN)) ;
```

Das "Aufräumen" geschieht in **CMainFrame::OnClose**:

```
void CMainFrame::OnClose ()
{
   KillTimer (1) ;
   delete m_penblue   ;
   delete m_penwhite  ;
   delete m_brushblue ;
   delete m_brushcyan ;
   CFrameWnd::OnClose () ;
}
```

In **CMainFrame::OnPaint** werden die Zeichenaktionen mit dem Initialisieren und dem Setzen der Transformation für die Anfangslage der Kurbel nur noch vorbereitet:

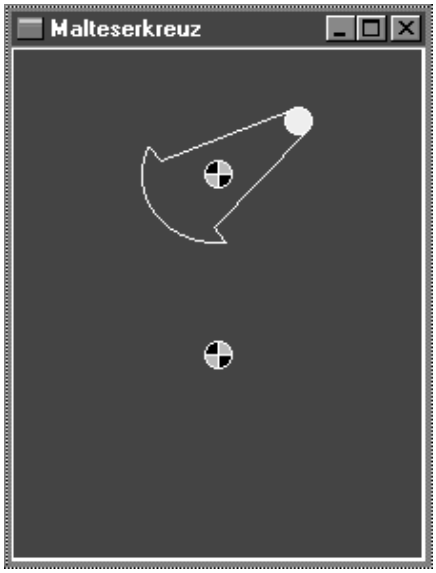
```
m_OmT  = - GI_PI / 2  ;                 // Anfangsstellung
m_dOmT =   GI_PI / 15 ;                 // Schrittweite
m_gi.tinit_gi () ;                      // Transformation initialisieren
m_gi.trota_gi (0. , m_L , m_OmT) ;      // Transformation setzen
```

Gezeichnet wird dagegen nur noch bei der Auswertung der Botschaft WM_TIMER in **CMainFrame::OnTimer**, jeweils zweimal: Zunächst wird mit der Hintergrundfarbe ("Pen" und "Brush") das alte Bild überzeichnet, danach wird der Winkel **m_OmT** (wird nur für die Transformationen in der Funktion **DrawCrank** benötigt) für die Rotations-Transformation um

das Inkrement **m_dOmT** vergrößert (wenn eine volle Umdrehung absolviert worden ist, wird **m_OmT** wieder um 2π verkleinert). Mit **CPT::trota_pt** wird eine **zusätzliche** Rotation um **m_dOmT** eingestellt, und das Bild wird neu gezeichnet:

```
void CMainFrame::OnTimer (UINT nIDTimer)
{
   CClientDC  dc (this) ;
   m_gi.setdc_gi (&dc)  ;
   DrawCrank (*m_penblue , *m_brushblue) ;            // Kurbel ueberzeichnen
   m_OmT += m_dOmT ;
   if (m_OmT > GI_PI * 1.5) m_OmT -= GI_PI * 2. ;
   m_gi.trota_pt (0. , m_L , m_dOmT) ;                // Zusaetzliche Drehung
   DrawCrank (*m_penwhite , *m_brushcyan) ;           // Kurbel neu zeichnen
   m_gi.remdc_gi () ;
}
```

Die nebenstehende Abbildung zeigt die Ausgabe des Programms **maltesr2.cpp** (auf dem Papier natürlich nur einen Schnappschuß).



Die Kurbel dreht sich

Hinweis: Hier sollen die ebenen Transformationsroutinen demonstriert werden, **nicht das Programmieren von Animationen**. Dafür sind noch ein paar zusätzlich "Tricks" sehr nützlich, die nicht Gegenstand der hier angestellten Betrachtungen sind. Aber die Transformationen sind natürlich ein ganz besonders wichtiges Hilfsmittel für das Programmieren von Animationen.

Das Programm **maltesr3.c** stellt schließlich sowohl die Kurbel als auch das Malteserkreuz in der Bewegung dar. Dabei sind zwei Probleme zu lösen:

- Die Synchronisation beider Bewegungen kann nur gelingen, wenn das kinematische Bewegungsgesetz des Malteserkreuz-Getriebes beachtet wird. Das Bewegungsgesetz kann man z. B. in "Dankert/Dankert: Technische Mechanik, computerunterstützt" auf Seite 449 finden. Es findet sich im Programm **maltesr3.c** in der Zeile (in **CMainFrame::OnTimer**), in der der Winkel **m_Phi**, mit dem die Rotation des Malteserkreuzes beschrieben wird, aus dem Winkel **m_OmT**, der die Stellung der Kurbel beschreibt, berechnet wird. Bei der Programmierung ist zu beachten, daß dieser Zusammenhang nur gilt, wenn die Kurbel mit dem Malteserkreuz im Eingriff steht, ansonsten bleibt **m_Phi** ungeändert.

- Es müssen zwei unterschiedliche Transformationen verwaltet werden (für Kurbel bzw. Malteserkreuz), jede Transformation wird einmal für das Zeichnen und (bei der folgenden WM_TIMER-Botschaft) für das Überzeichnen der alten Lage benötigt. Um die Transformationsmatrizen nicht stets neu berechnen zu müssen, werden in **maltesr3.cpp** die Methoden **CPT::tgttm_pt** und **CPT::tsttm_pt** verwendet, mit denen die aktuelle Transformationsmatrix erfragt bzw. gesetzt wird. Eine andere Möglichkeit, das Problem der Verwaltung mehrerer erforderlicher Transformationen zu lösen, wird in der endgültigen Programm-Variante **malteser.cpp** vorgestellt.

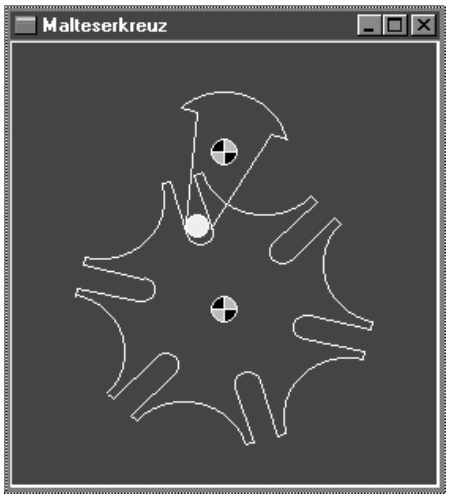
Bei der Auswertung von WM_PAINT in **CMainFrame::OnPaint** werden die Anfangswerte für die Transformationsmatrizen gesetzt und mit **CPT::tgttm_pt** auf die Felder **m_tmcross** bzw. **m_tmcrank** (gehören zur Klasse **CMainFrame**, müssen jeweils 9 **double**-Werte aufnehmen können) gesichert:

```
void CMainFrame::OnPaint ()
{
   CPaintDC  dc (this) ;
   m_gi.init_gi (this , &dc) ;
   m_gi.vback_gi (*m_brushblue , *m_penwhite) ;
   m_gi.stuci_gi (- m_L , - m_L , m_L , m_L + m_R + m_Rs , 5.) ;
   DrawPivot (0.)   ;                  // Lagersymbole
   DrawPivot (m_L)  ;                  // zeichnen
   m_Phi  =   0.          ;
   m_OmT  = - GI_PI /  2 ;             // Anfangsstellung
   m_dOmT =   GI_PI / 15 ;             // Schrittweite
   m_gi.tinit_pt () ;                  // Transformation initialisieren
   m_gi.tgttm_pt (m_tmcross) ;         // = Start-Transformation fuer Kreuz
   m_gi.trota_pt (0. , m_L , m_OmT) ;  // Transformation setzen
   m_gi.tgttm_pt (m_tmcrank) ;         // ... und sichern
   m_gi.remdc_gi () ;
}
```

Bei der Auswertung der Botschaft WM_TIMER in **CMainFrame::OnTimer** wird dann jeweils die passende Transformationsmatrix gesetzt (mit **CPT::tsttm_pt**), die natürlich entsprechend der Änderungen der Winkel verändert (und mit **CPT::tgttm_pt** nach jeder Änderung gesichert) werden muß. Für die Kurbel wird die Änderung wie in **maltesr2.cpp** inkrementell mit **CPT::trota_pt** realisiert, die Transformation für das Malteserkreuz wird mit **CPT::ttabs_pt** "gesetzt" (ohne Berücksichtigung vorheriger Transformationen, mit **CPT::ttabs_pt** können gleichzeitig eine Skalierung bezüglich des Nullpunktes, eine Rotation um den Nullpunkt und eine Translation als neue "Initialisierung" der Transformationsmatrix erzeugt werden):

```
void CMainFrame::OnTimer (UINT nIDTimer)
{
   CClientDC  dc (this) ;
   m_gi.setdc_gi (&dc)  ;
   m_gi.tsttm_pt (m_tmcrank) ;                  // Transformation fuer Kurbel
   DrawCrank (*m_penblue , *m_brushblue) ;      // Kurbel ueberzeichnen
   m_gi.trota_pt (0. , m_L , m_dOmT) ;          // Transformation aendern
   m_gi.tgttm_pt (m_tmcrank) ;                  // ... und sichern
   m_gi.tsttm_pt (m_tmcross) ;                  // Transformation fuer Kreuz
   m_OmT += m_dOmT ;
   if (m_OmT > GI_PI * 1.5) m_OmT -= GI_PI * 2. ;
   if (m_OmT > - GI_PI / 3. && m_OmT < GI_PI / 3.)   // Kurbel im Eingriff
   {
      m_Phi = - atan2 (sin (m_OmT) , m_L / m_R - cos (m_OmT)) - GI_PI / 6. ;
      DrawCross (*m_penblue) ;                   // Kreuz ueberzeichnen
   }
   m_gi.tsttm_pt (m_tmcrank) ;                  // Transformation fuer Kurbel
   DrawCrank (*m_penwhite , *m_brushcyan) ;     // Kurbel neu zeichnen
   m_gi.ttabs_pt (0. , 0. , m_Phi , 1. , 1. ) ; // Transformation fuer Kreuz
   DrawCross (*m_penwhite) ;                    // Kreuz neu zeichnen
   m_gi.tgttm_pt (m_tmcross) ;                  // Transformation sichern
   m_gi.remdc_gi () ;
}
```

Damit wird die Bewegung des gesamten Malteserkreuz-Getriebes simuliert, das eine kontinuierliche Drehbewegung (Kurbel) in eine schrittweise Bewegung (Malteserkreuz) umwandelt.



Kurbel und Kreuz rotieren synchron

In der endgültigen Programm-Version **malteser.cpp**, deren Ausgabe sich von der Ausgabe von **maltesr3.cpp** nicht unterscheidet, wird eine alternative Möglichkeit demonstriert, mehrere Transformationen zu verwalten: Es werden **zwei CGI-Objekte** erzeugt (für jedes bewegte Teil eins), die jeweils eigene Transformationsmatrizen verwalten:

```
class CMainFrame : public CFrameWnd
{
  private:
     CGI  m_gi1 ;  // ... fuer Malteserkreuz
     CGI  m_gi2 ;  // ... fuer Kurbel
  // ...
} ;
```

In **CMainFrame::OnPaint** muß nun einiges doppelt erledigt werden (Initialisieren der **CGI**-Objekte, Einstellen der "User coordinates" mit **CGI::stuci_gi**, Initialisieren der Transformationen mit **CPT:tinit_pt**). Dafür entfällt das Sichern der Transformationsmatrizen.

Für einige Aktionen, die ohne Bezug auf die Transformationen ablaufen, kann ein beliebiges **CGI**-Objekt benutzt werden (z. B. für das Zeichnen des Hintergrundes und für das Zeichnen der Lagersymbole in **DrawPivot**). In **DrawCross** wird dagegen konsequent mit **m_gi1** gearbeitet, in **DrawCrank** mit **m_gi2**.

Die Zeichenaktionen in **OnTimer** werden deutlich vereinfacht, weil die Verwaltung der unterschiedlichen Transformationsmatrizen entfällt:

```
void CMainFrame::OnTimer (UINT nIDTimer)
{
   CClientDC  dc (this) ;
   m_gi1.setdc_gi (&dc)  ;
   m_gi2.setdc_gi (&dc)  ;
   DrawCrank (*m_penblue , *m_brushblue) ;       // Kurbel ueberzeichnen
   m_OmT += m_dOmT ;
   if (m_OmT > GI_PI * 1.5) m_OmT -= GI_PI * 2. ;
   if (m_OmT > - GI_PI / 3. && m_OmT < GI_PI / 3.)   // Kurbel im Eingriff
   {
      m_Phi = - atan2 (sin (m_OmT) , m_L / m_R - cos (m_OmT)) - GI_PI / 6. ;
      DrawCross (*m_penblue) ;                    // Kreuz ueberzeichnen
   }
   m_gi2.trota_pt (0. , m_L , m_dOmT) ;          // Transformation aendern
   DrawCrank (*m_penwhite , *m_brushcyan) ;      // Kurbel neu zeichnen
   m_gi1.ttabs_pt (0. , 0. , m_Phi , 1. , 1. ) ; // Transformation fuer Kreuz
   DrawCross (*m_penwhite) ;                     // Kreuz neu zeichnen
   m_gi1.remdc_gi () ;
   m_gi2.remdc_gi () ;
}
```

Welche der beiden Varianten zu bevorzugen ist, kann kaum generell entschieden werden. Für komplizierte Probleme ist die in **malteser.cpp** realisierte Strategie für den Programmierer wohl übersichtlicher.